ARTICLE IN PRESS

Computer Physics Communications ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

A GPU-enabled Finite Volume solver for global magnetospheric simulations on unstructured grids

Andrea Lani [a,*], Mehmet Sarp Yalim [b], Stefaan Poedts [b]

[a] Von Karman Institute for Fluid Dynamics, Waterloosesteenweg 72, 1640, Sint Genesius Rode, Belgium
[b] KU Leuven/Centrum voor mathematische Plasma-Astrofysica, Celestijnenlaan 200B, B-3001 Leuven, Belgium

ARTICLE INFO

Article history:
Received 8 April 2014
Received in revised form
2 June 2014
Accepted 4 June 2014
Available online xxxx

Keywords:
Magnetohydrodynamics
Finite Volume
GPU
Object-oriented

ABSTRACT

This paper describes an ideal Magnetohydrodynamics (MHD) solver for global magnetospheric simulations based on a $\mathbf{B_1} + \mathbf{B_0}$ splitting approach, which has been implemented within the COOLFluiD platform and adapted to run on modern heterogeneous architectures featuring General Purpose Graphical Processing Units (GPGPUs). The code is based on a state-of-the-art Finite Volume discretization for unstructured grids and either explicit or implicit time integration, suitable for both steady and time accurate problems. Innovative object-oriented design and coding techniques mixing C++ and CUDA are discussed. Performance results of the modified code on single and multiple processors are presented and compared with those provided by the original solver.

0. Introduction

Complex multi-physics scientific problems need to be simulated in parallel on large High Performance Computing (HPC) clusters in order to provide solutions within reasonable execution times. The recent advent of general-purpose accelerators like Graphics Processing Units (GPUs) has contributed to move the state-of-the-art of scientific computing a step forward, providing significant speedups (up to a factor of 10 or more) for several applications. Hybrid architectures combining multi-core CPUs and multi-core GPUs are emerging and simulation tools have slowly started to evolve to take profit of the greatly enhanced computational capabilities. Prediction codes for heliospheric physics and Space Weather phenomena, including the modeling of Coronal Mass Ejections and the interaction of the solar wind with planetary magnetospheres, ideally requiring real-time or even faster-than-real-time solutions, typically involve intensive floating point operations and complex parallelizable algorithms, which can take considerable benefit from the newly available GPU-based technology. Global magnetospheric simulations typically rely on single-fluid ideal Magnetohydrodynamics (MHD) models with Finite Volume (FV) or Finite Difference algorithms, at best featuring Adaptive Mesh Refinement (AMR) to better resolve the smallest scales.

Some pioneering attempts to port global MHD algorithms on GPUs are reported in the literature [1,2], showing promising results even on large scale simulations. To the Authors' knowledge, all of those efforts have been applied to solvers for structured grids using explicit time stepping (e.g. Runge–Kutta schemes) and have never addressed any code design issues within a modern object-oriented infrastructure.

This paper will, instead, discuss the details and performance of the porting to GPU of a parallel unstructured 3D ideal MHD solver [3,4] for Space Weather applications. The code has been implemented within the Computational Object-Oriented Libraries for Fluid Dynamics (COOLFluiD) platform [5–8]. It can tackle both steady and time-accurate problems and is based on a state-of-the-art FV algorithm. Both explicit and implicit time stepping procedures are supported and discussed here.

All the developments have been based on a combination of C++ and the Compute Unified Device Architecture (CUDA), i.e. NVIDIA's parallel computing platform. The main effort has consisted in interfacing CUDA C calls transparently into the fast-path-code (i.e. the most frequently called routines mostly performing array-based linear algebra) and in optimizing the data transfer between CPU and GPU (and vice versa), while keeping the high level solver structure (mesh decomposition, space–time discretization of MHD equations, etc.)

* Corresponding author.
E-mail addresses: lani@vki.ac.be (A. Lani), MehmetSarp.Yalim@wis.kuleuven.be (M.S. Yalim), Stefaan.Poedts@wis.kuleuven.be (S. Poedts).

http://dx.doi.org/10.1016/j.cpc.2014.06.001
0010-4655/

formally untouched. The whole porting has benefited from the object-oriented and modular design of COOLFluiD, where different levels of abstraction coexist and data access is encapsulated, so that the application code does not see either the internal data representation or its parallel access pattern, which is a key feature for keeping transparent GPU computing with CUDA. Performance results of the modified code are also discussed and compared with those obtained with the original fully CPU-based solver.

0.1. COOLFluiD platform

COOLFluiD [5,6,9,10] is a collaborative software environment for high-performance scientific computing where different numerical techniques, physical models, post-processing algorithms can coexist and work together. Herein, each numerical method or physical model is encapsulated into an independent dynamic module (or *plug-in* library) that can be loaded on demand by user-defined applications. Some of the main features of COOLFluiD include: parallel solvers for compressible and incompressible flows based on multiple discretization techniques (e.g. FV, Residual Distribution schemes, Finite Element, Spectral Finite Differences) for unstructured meshes, interfaces to different linear systems solvers (e.g. PETSc, Trilinos, Pardiso), aerothermochemical models for flows and plasma [11–15], MHD, coupling algorithms for multi-physics and multi-domain simulations, Arbitrary Lagrangian Eulerian methods and radiation transport algorithms based on Monte Carlo [8,16].

1. COOLFluiD MHD solver

This section describes the parallel explicit/implicit, steady/unsteady cell-centered FV solver for ideal MHD equations which has been ported to GPUs within the present project.

1.1. Governing equations

The plasma behavior is modeled by the system of compressible ideal MHD equations. The latter are formed by a coupling of Euler and Maxwell's equations representing the hydrodynamic behavior and magnetic environment of the plasma, respectively. They consist of 8 partial differential equations (PDEs) for conservation of mass, momentum and energy of plasmas together with the induction equation for the magnetic field. Furthermore, an additional equation (i.e. $\nabla \cdot \mathbf{B} = 0$) should always be satisfied. This equation is called the *solenoidal constraint*. It is challenging to satisfy the solenoidal constraint at numerical discretization level. Therefore, there are various techniques proposed in the literature to satisfy it up to different accuracy levels [17,18]. For our target applications focusing on interaction between solar wind and Earth magnetosphere, the magnitude of the magnetic field of the plasma varies over a wide range and hence involves large gradients especially in the vicinity of the Earth. This can cause the occurrence of negative pressure values in such regions and therefore the breakdown of simulations. For this purpose, a special treatment known as the $\mathbf{B_0} + \mathbf{B_1}$ splitting (or Magnetic Field Splitting (MFS)) technique of [19] is applied to the governing equations, which are modified using the hyperbolic divergence cleaning (HDC) approach with hyperbolic Lagrange multiplier [18] to satisfy the solenoidal constraint. The resulting system of PDEs written in non-dimensional, conservative and differential form can be written as:

$$\frac{\partial \mathbf{U}}{\partial \mathbf{P}} \frac{\partial \mathbf{P}}{\partial t} + \nabla \cdot \mathbf{F}_{HDC-MFS} + \nabla \cdot \mathbf{F}_{HDC-Rest} = 0, \tag{1}$$

where $\frac{\partial \mathbf{U}}{\partial \mathbf{P}}$ is the transformation matrix between conservative variables $\mathbf{U}$ and primitive variables $\mathbf{P}$, which are defined as:

$$\mathbf{U} = \begin{pmatrix} \rho \\ \rho \mathbf{V} \\ \mathbf{B} \\ E_1 = E - \mathbf{B}_1 \cdot \mathbf{B}_0 - B_0^2/2 \\ \phi \end{pmatrix}, \qquad \mathbf{P} = \begin{pmatrix} \rho \\ \mathbf{V} \\ \mathbf{B} \\ p \\ \phi \end{pmatrix}. \tag{2}$$

Herein, ρ is the density, $\mathbf{V}$ is the velocity, $\mathbf{B} = \mathbf{B_0} + \mathbf{B_1}$ is the total magnetic field, p is the pressure, and ϕ is a scalar potential function that constrains $\mathbf{B}$ to the space of divergence free fields [20,4]. While $\mathbf{B_1}$ denotes the variable magnetic field component to be solved for, $\mathbf{B_0}$ represents a constant Earth's dipole magnetic field which is given by:

$$\mathbf{B_0} = \frac{1}{r^3}\left(3(\mathbf{m} \cdot \mathbf{n_r})\mathbf{n_r} - \mathbf{m}\right), \tag{3}$$

where $\mathbf{m}$ is the Earth's magnetic dipole moment, $\mathbf{n_r}$ is a unit vector in the $\mathbf{r}$-direction, and r denotes the distance from the center of the dipole chosen as the origin. This magnetic field is a potential field and satisfies the three conditions:

$$\frac{\partial \mathbf{B_0}}{\partial t} = \nabla \cdot \mathbf{B_0} = \nabla \times \mathbf{B_0} = 0. \tag{4}$$

Moreover, the energy per unit volume E_1 is given by:

$$E_1 = \frac{p}{\gamma - 1} + \frac{1}{2}\left(\rho V^2 + B_1^2\right). \tag{5}$$

where γ is the ratio of specific heats. The plasma is assumed to obey the ideal gas law and to be calorically perfect (i.e. the specific heats and, therefore, γ are constant). γ is assumed to be $5/3$ in our case. The formulation in Eq. (1) allows for computing, extrapolating, limiting and updating the solution in terms of primitive variables instead of conservative variables. This is especially advantageous because, by

Please cite this article in press as: A. Lani, et al., A GPU-enabled Finite Volume solver for global magnetospheric simulations on unstructured grids, Computer Physics Communications (2014), http://dx.doi.org/10.1016/j.cpc.2014.06.001

offering an easy way to direct controlling over the pressure, it becomes easy to improve stability especially in early or critical stages of the computation when pressure tends to go negative.

The convective fluxes $\mathbf{F}^c$ are split into a part (HDC-MFS) containing only the unknown $\mathbf{B_1}$ and a remaining part (HDC-Rest) depending on both $\mathbf{B_1}$ and $\mathbf{B_0}$ [19,21]:

$$\mathbf{F}_{HDC-MFS} = \begin{pmatrix} \rho\mathbf{V} \\ \rho\mathbf{V}\mathbf{V} + (p + B_1^2/2)\bar{\mathbf{I}} - \mathbf{B}_1\mathbf{B}_1 \\ \mathbf{V}\mathbf{B}_1 - \mathbf{B}_1\mathbf{V} + \bar{\mathbf{I}}\phi \\ (E_1 + p + B_1^2/2)\mathbf{V} - (\mathbf{V}\cdot\mathbf{B}_1)\mathbf{B}_1 \\ V_{ref}^2\mathbf{B}_1 \end{pmatrix} \tag{6}$$

$$\mathbf{F}_{HDC-Rest} = \begin{pmatrix} 0 \\ (\mathbf{B}_0\cdot\mathbf{B}_1)\bar{\mathbf{I}} - (\mathbf{B}_0\mathbf{B}_1 + \mathbf{B}_1\mathbf{B}_0) \\ \mathbf{V}\mathbf{B}_0 - \mathbf{B}_0\mathbf{V} \\ (\mathbf{B}_0\cdot\mathbf{B}_1)\mathbf{V} - (\mathbf{V}\cdot\mathbf{B}_1)\mathbf{B}_0 \\ 0 \end{pmatrix}, \tag{7}$$

where, in particular, V_{ref} is a reference speed with which $\nabla\cdot\mathbf{B}$ errors are radiated away in all directions. The value of V_{ref} is case-dependent.

1.2. Space discretization

The COOLFluiD MHD code is a parallel FV solver for unstructured grids. The FV discretization is applied to the system of governing equations written in integral conservation form:

$$\frac{\partial}{\partial t}\int_{\Omega}\mathbf{U}\,d\Omega + \oint_{d\partial\Omega}\mathbf{F}^c\cdot\mathbf{n}\,d\partial\Omega = 0, \tag{8}$$

where $\mathbf{F}^c = \mathbf{F}_{HDC-MFS} + \mathbf{F}_{HDC-Rest}$. We apply a conventional cell-centered approximation, which assumes solution vectors located at the centroid of each computational cell. Inverse-distance weighted least square reconstruction [22] is utilized to yield second order accuracy. Oscillation free solutions are obtained with Barth–Jespersen's multidimensional limiter [23].

1.2.1. Linear least squares reconstruction

On a general polyhedral unstructured mesh, the cell-wise gradient ∇u can be computed with a least square (LS) approach as the result of the following linear system [22]:

$$[\mathbf{L}_x\ \ \mathbf{L}_y\ \ \mathbf{L}_z]\ \ \nabla u_i = \mathbf{f}_u. \tag{9}$$

The matrix on the LHS is generally non-square and its column vectors $\mathbf{L}_d$ are defined as:

$$\mathbf{L}_d = [w_1(\Delta x_d)_1, \ldots, w_{N_i}(\Delta x_d)_{N_i}]^T, \tag{10}$$

where the weights w_k multiply the distances between the centroid of the current cell and the centroids of its N_i neighbor cells, belonging to the chosen computational stencil. Linear weights can be based on the inverse of distances and computed as $w_j = 1/\parallel \boldsymbol{\Delta x}_j \parallel$.

The non linear RHS vector $\mathbf{f}_u$ reads:

$$\mathbf{f}_u = [w_1\Delta(u_1 - u_i), \ldots, w_N(u_{N_i} - u_i)]^T. \tag{11}$$

The system in Eq. (9) can be solved in a least squares sense with an orthogonalization technique, leading to:

$$[\mathbf{L}_x\ \ \mathbf{L}_y\ \ \mathbf{L}_z]^T\cdot[\mathbf{L}_x\ \ \mathbf{L}_y\ \ \mathbf{L}_z]\ \ (\nabla u)_{\Omega_i} = [\mathbf{L}_x\ \ \mathbf{L}_y\ \ \mathbf{L}_z]^T\cdot\mathbf{f}_u. \tag{12}$$

After having defined the dot products $l_{jk} = \mathbf{L}_j\cdot\mathbf{L}_k$ and $f_j = \mathbf{L}_j\cdot\mathbf{f}_u$, Eq. (12) simplifies to:

$$\nabla u_i = \{\bar{\mathbf{l}}_{jk}\}^{-1}\mathbf{f}_l \tag{13}$$

where

$$\{\bar{\mathbf{l}}_{jk}\} = [\mathbf{L}_x\ \ \mathbf{L}_y\ \ \mathbf{L}_z]^T\cdot[\mathbf{L}_x\ \ \mathbf{L}_y\ \ \mathbf{L}_z], \qquad \mathbf{f}_l = [fx,\ \ fy,\ \ fz]^T. \tag{14}$$

If we take into account the definition of the inverse for a 3×3 matrix, Eq. (13) can be developed further and gives:

$$\nabla u_i = \frac{1}{\det(\{\bar{\mathbf{l}}_{jk}\})}\begin{pmatrix} \det(\bar{\mathbf{M}}_{xx}^L)f_x + \det(\bar{\mathbf{M}}_{xy}^L)f_y + \det(\bar{\mathbf{M}}_{xz}^L)f_z \\ \det(\bar{\mathbf{M}}_{xy}^L)f_x + \det(\bar{\mathbf{M}}_{yy}^L)f_y + \det(\bar{\mathbf{M}}_{yz}^L)f_z \\ \det(\bar{\mathbf{M}}_{xz}^L)f_x + \det(\bar{\mathbf{M}}_{yz}^L)f_y + \det(\bar{\mathbf{M}}_{zz}^L)f_z \end{pmatrix} \tag{15}$$

where $\bar{\mathbf{M}}_{jk}^L$ is a minor of the matrix $\{\bar{\mathbf{l}}_{jk}\}$. The system in Eq. (12) is not necessarily well posed and a sufficiently large stencil is needed to avoid singularities ($\det(\{\bar{\mathbf{l}}_{jk}\}) \simeq 0$). In our case, we consider all distance-1 neighbors (i.e. all cell vertex neighbors) as part of the reconstruction stencil (see Fig. 1), since this choice provides the best compromise between numerical accuracy and stability [24,25,10].

Please cite this article in press as: A. Lani, et al., A GPU-enabled Finite Volume solver for global magnetospheric simulations on unstructured grids, Computer Physics Communications (2014), http://dx.doi.org/10.1016/j.cpc.2014.06.001

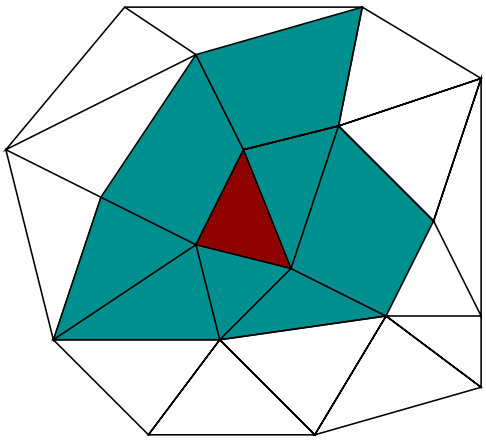

Fig. 1. Full stencil used for the linear least squares algorithm and the flux limiter.

1.2.2. Barth–Jespersen's limiter

The formula for Barth–Jespersen's limiter [23] reads:

$$\phi_{\mathrm{j,P}} = \begin{cases} \min\left(1, \dfrac{\mathbf{P}_{\max\,\mathrm{j\&ng}} - \mathbf{P}_{\mathrm{j}}}{\nabla\mathbf{P}|_{\mathrm{j}} \cdot \left(\mathbf{r}_{\mathrm{P}} - \mathbf{r}_{\mathrm{j}}\right)}\right) & \text{if } \nabla\mathbf{P}|_{\mathrm{j}} \cdot \left(\mathbf{r}_{\mathrm{P}} - \mathbf{r}_{\mathrm{j}}\right) > 0, \\ \min\left(1, \dfrac{\mathbf{P}_{\min\,\mathrm{j\&ng}} - \mathbf{P}_{\mathrm{j}}}{\nabla\mathbf{P}|_{\mathrm{j}} \cdot \left(\mathbf{r}_{\mathrm{P}} - \mathbf{r}_{\mathrm{j}}\right)}\right) & \text{if } \nabla\mathbf{P}|_{\mathrm{j}} \cdot \left(\mathbf{r}_{\mathrm{P}} - \mathbf{r}_{\mathrm{j}}\right) < 0, \\ 1 & \text{if } \nabla\mathbf{P}|_{\mathrm{j}} \cdot \left(\mathbf{r}_{\mathrm{P}} - \mathbf{r}_{\mathrm{j}}\right) = 0 \end{cases} \tag{16}$$

where $\mathbf{P}_{\max\,\mathrm{j\&ng}}$ and $\mathbf{P}_{\min\,\mathrm{j\&ng}}$ are the maximum and minimum average values among the cell, j, and all its vertex neighbors, i.e. considering the same stencil as for the least squares reconstruction. As limiter value for each variable in a cell, j, the minimum limiter value corresponding to that variable in all the quadrature points of that cell is taken. The limiter is applied to primitive variables in our case.

1.2.3. Local Lax–Friedrichs scheme

A modified version of the local Lax–Friedrichs (a.k.a. Rusanov) flux [26,20] is used for computing the numerical convective fluxes on each interface f between cells left L and right R. It can be written as:

$$\mathbf{F}_f^{LF} = \frac{1}{2}\left(\mathbf{F}(\mathbf{U}^R) + \mathbf{F}(\mathbf{U}^L)\right) - \frac{1}{2}k\,|\lambda|\left(\mathbf{U}^R - \mathbf{U}^L\right), \tag{17}$$

where $\mathbf{F}^*$ are the physical fluxes, k is a coefficient for reducing numerical dissipation, such that $0 < k \leq 1$, and λ is the maximum eigenvalue. During a simulation, the factor k can be reduced (either interactively or with a prescribed analytical law) up to a value as low as 0.1, giving solutions as accurate as a Roe scheme [27] would give, as reported in [4]. For the simulations performed in this paper where we use the second-order accurate TVD Rusanov scheme, k is assigned to be 1.

1.2.4. Implicit time integration

The discretized time-dependent system in Eq. (8) is re-written as:

$$\tilde{\mathbf{R}}(\mathbf{P}) = \frac{\partial\mathbf{U}}{\partial\mathbf{P}}\frac{\partial\mathbf{P}}{\partial t} + \mathbf{R}(\mathbf{P}) = \mathbf{0}, \tag{18}$$

where $\tilde{\mathbf{R}}(\mathbf{P})$ is an array of *pseudo-steady* residuals, $\mathbf{U}$ are conservative variables, $\mathbf{P}$ are *update* variables (e.g. conservative or primitive in our case). The vectors of the conservative and primitive variables are given in Eq. (2). The application of a Newton linearization, for each sub-iteration step (one in steady cases, more in time-accurate cases) yields linear systems of the type:

$$\begin{cases} \left[\dfrac{\partial\tilde{\mathbf{R}}}{\partial\mathbf{P}}\left(\mathbf{P}^k\right)\right]\Delta\mathbf{P}^k = -\tilde{\mathbf{R}}(\mathbf{P}^k) \\ \mathbf{P}^{k+1} = \mathbf{P}^k + \Delta\mathbf{P}^k \end{cases}, \tag{19}$$

where the jacobian matrix $\frac{\partial\tilde{\mathbf{R}}}{\partial\mathbf{P}}$ is computed numerically. The definition of $\tilde{\mathbf{R}}(\mathbf{P})$ depends on the chosen steady or unsteady time integration scheme. We use the following definitions:

$$\begin{aligned} \tilde{\mathbf{R}}(\mathbf{P}) &= \frac{\mathbf{U}(\mathbf{P}) - \mathbf{U}(\mathbf{P}^k)}{\Delta t}\Omega + \mathbf{R}(\mathbf{P}) \quad \text{Backward Euler} \\ \tilde{\mathbf{R}}(\mathbf{P}) &= \frac{\mathbf{U}(\mathbf{P}) - \mathbf{U}(\mathbf{P}^k)}{\Delta t}\Omega + \frac{1}{2}[\mathbf{R}(\mathbf{P}) + \mathbf{R}(\mathbf{P}^k)] \quad \text{Crank–Nicholson} \\ \tilde{\mathbf{R}}(\mathbf{P}) &= \frac{3\mathbf{U}(\mathbf{P}) - 4\mathbf{U}(\mathbf{P}^k) + \mathbf{U}(\mathbf{P}^{k-1})}{2\Delta t}\Omega + \mathbf{R}(\mathbf{P}) \quad \text{3-Point Backward.} \end{aligned} \tag{20}$$

Please cite this article in press as: A. Lani, et al., A GPU-enabled Finite Volume solver for global magnetospheric simulations on unstructured grids, Computer Physics Communications (2014), http://dx.doi.org/10.1016/j.cpc.2014.06.001

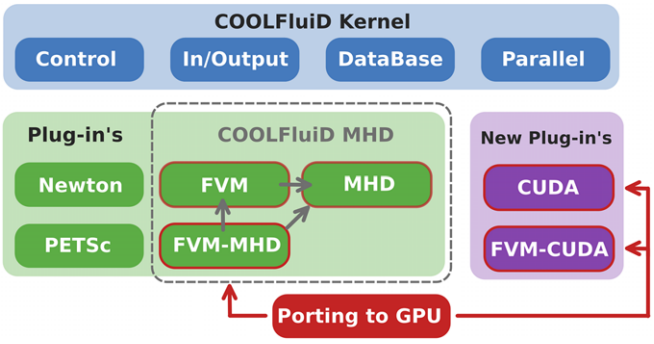

Fig. 2. Schematics of the COOLFluiD MHD solver before and after its extension to GPUs.

GMRES algorithms complemented with parallel preconditioners (e.g. Additive Schwartz method, block Jacobi) provided by the PETSc toolkit [28] are used to solve the corresponding linear systems arising from Newton linearizations.

The solution update is also performed in primitive variables. Herein, in a steady case, the solution at the new time step $\mathbf{P}^{n+1}$ can be directly set equal to $\mathbf{P}^{k+1}$ after one Newton sub-iteration. In an unsteady case, we must keep on iterating till $\|\Delta\mathbf{P}^{k}\| < \varepsilon$ and then set $\mathbf{P}^{n+1} = \mathbf{P}^{k_{last}+1}$.

2. Porting to GPU

The code design is sketched in Fig. 2. The COOLFluiD kernel provides the high-level control of the simulation, parallel data-structure and I/O capabilities, a set of basic interfaces for models and algorithms. The implementation of such interfaces is provided by objects which are organized in plug-in libraries. The core of COOLFluiD MHD includes:

- the MHD module featuring the governing equations (convective fluxes, variable sets, transformations);
- the FVM module implementing a generic FV algorithm;
- the FVM–MHD module collecting some FV schemes and boundary conditions tailored to MHD.

Other useful libraries which are dynamically linked to the main solver are:

- the RK module (not shown in the picture) implementing Runge–Kutta explicit time stepping;
- the Newton iterator module implementing implicit time integrators;
- the interface to the PETSc toolkit which is used to solve linear systems.

The porting to GPUs has been based on the CUDA C/C++ language from NVIDIA. To this end, two new modules have been created: CUDA and FVM–CUDA. While trying to reuse as much as possible of the existing solver, only a few additional adaptations in MHD, FVM and FVM–MHD modules were needed to port the solver to GPU.

2.1. CUDA module

The CUDA module, associated to a namespace *CudaEnv* and integrated into the COOLFluiD kernel, includes some reusable tools such as:

- `CudaDeviceManager`, a singleton interface to initialize the CUDA environment and hold information about number of blocks and threads to be used on the device;
- `CudaVector`, a proxy interface for handling (creating, copying, deleting) CPU/GPU arrays and data transfers from/to device (GPU–CPU and CPU–GPU), partially inspired to [29];
- a customized implementation of vectors and matrices using GPU-enabled expression templates.

As far as `CudaDeviceManager` is concerned, while the number of threads per block `NTHREADS_PER_BLOCK` is user-defined in our simulations (a value of 32 give the best performance in our case), the number of blocks for the kernel is computed as shown in code listing 1. Herein, `N` is the local number of cells in the mesh partition and `NBLOCKS` is chosen as the maximum number of blocks allowed on the device.

Listing 1: CudaDeviceManager::getBlocksPerGrid() function

```
int getBlocksPerGrid (int N)
{
   return min(NBLOCKS, (N + NTHREADS_PER_BLOCK-1) / NTHREADS_PER_BLOCK);
}
```

2.1.1. CudaVector

Listing 2: CudaVector class definition

```
// ---- CudaVector.hh ---- //
template <typename T, template <typename T1 = T> class ALLOC = PinnedHostAlloc>
class CudaVector {
public:
 CudaVector();                        // default constructor
 CudaVector (const CudaVector& in); // copy constructor
```

Please cite this article in press as: A. Lani, et al., A GPU-enabled Finite Volume solver for global magnetospheric simulations on unstructured grids, Computer Physics Communications (2014), http://dx.doi.org/10.1016/j.cpc.2014.06.001

```
/// Constructor for creating logically 2D arrays
/// @param init      value to use for initialization
/// @param ns        full size on the CPU
/// @param stride    stride (>=1)
CudaVector(T init, size_t ns, size_t stride);

/// assignment operator (from input array)
template <typename INPUT> const CudaVector& operator=(const INPUT& in);

/// assignment operator (from input scalar value)
const CudaVector& operator=(const T& value);

~CudaVector(); // destructor

void put();  // synchronous copy of all data from CPU to GPU
void get();  // synchronous copy of all data from GPU to CPU
void free(); // free the memory

T* ptr();                       // access raw pointer to CPU array
T* ptrDev();                    // access raw pointer to GPU array
T operator[] (size_t i) const; // subscripting operator for accessing CPU data
T& operator[] (size_t i);      // subscripting operator for accessing CPU data
T operator() (size_t i);       // access data on CPU taking stride into account
T& operator() (size_t i);      // access data on CPU taking stride into account

size_t size() const;                // size of the whole storage on the CPU
size_t stride() const;              // get the stride{return m_stride;}

private:
 ALLOC<T> m_alloc;   // allocator
 size_t   m_stride;  // stride (1 by default)

}; // end class CudaVector
```

`CudaVector` encapsulates all CUDA-specific functions for creating, copying, deleting arrays. Part of its C++ class definition is shown in code listing 2. Unlike *thrust* vectors [29], a `CudaVector` stores both a CPU (host) and a GPU (device) array according to the selected allocator (template parameter `ALLOC`). The host array can be allocated either on the pinned memory by `PinnedHostAlloc` or with *malloc* by `MallocHostAlloc`. We have used pinned memory for arrays needing to be transferred from/to device more than once, since the transfer is about 2X faster. Host/device arrays can be conveniently accessed via the functions *ptr*() and *ptrDev*(), while *put*() and *get*() will copy data to/from device. Logically 2D arrays can be created by specifying a stride > 1. Our implementation of `CudaVector` also includes the possibility of overlapping computation and data transfer by accessing array slices corresponding to multiple CUDA streams. However, since in our FV code data transfer time proved to be negligible (see Table 2) if compared to computational time, we did not use this optimization feature and we will not discuss the details here.

2.1.2. GPU-enabled expression templates

When dealing with FV schemes, it is common to deal with expressions involving algebraic operations among arrays, matrices and/or constants, such as in the Lax–Friedrichs flux that we recall here (from Section 1.2.3):

$$\mathbf{F}_f^{LF} = \frac{1}{2}\left(\mathbf{F}(\mathbf{U}^R) + \mathbf{F}(\mathbf{U}^L)\right) - \frac{1}{2}k\,|\lambda|\left(\mathbf{U}^R - \mathbf{U}^L\right). \tag{21}$$

In order to preserve both this vectorial formalism and the efficiency of hand-coded loops in a C++ implementation, Veldhuizen [30,31] and Vandervoorde [32] developed a powerful technique called Expression Templates (ET) in the mid nineties. Since then, a countless number of libraries (e.g. [33–36]) have been developed, contributing to extend the ET concept far beyond the original intent. The technique makes use of template meta-programming to implement mathematical expressions efficiently and to overcome all deficiencies provided by conventional operator overloading. The key idea consists in encoding generic expressions in template arguments and delaying the actual evaluation till the result assignment takes place.

Since none of the currently available open source libraries was providing an ET implementation of device arrays allowing linear symbolic algebra to work also within GPU kernels (even though there is some work in progress from [33] in this direction), a pre-existing ET implementation in COOLFluiD [37] was adapted to run on the device for this work. A brief overview of some key aspects of the resulting implementation is given hereafter.

ExtrT: base expression class. The core ingredients of our ET engine are a trait class `ETuple` and an expression class `ExprT` which are shown in code listing 3. In this case, a technique known as *Curiously Recurring Template Pattern* (CRTP) [32] is employed. The latter allows the class `ExprT` to delegate the actual evaluation to a generic derived class, corresponding to the first template parameter `DERIVED`. The second template parameter `ARG` provides information about the entry type and the static size of the deriving array classes, i.e. the same type information provided by `ETuple`. At the time of the writing, in order to make the ET mechanism work properly on the device with the CUDA compiler *nvcc*, it was necessary to replace the usage of C++ references "&" with pointers in declaring/accessing the member data `m_exdata`.

Listing 3: ExprT class definition

```
// —— ExprT.hh —— //

/// trait class
template <typename T, int N, int M>
```

```
struct ETuple {
  typedef T TYPE; // typedef allowing for accessing the type T
  enum {SIZE1=N}; // enum allowing for accessing the size N
  enum {SIZE2=M}; // enum allowing for accessing the size M
                  // M=0 for vectors, M>0 for matrices
};

/// expression class
template <typename DERIVED, typename ARG>
class ExprT {
public:
  typedef DERIVED PTR;      // typedef allowing for accessing DERIVED
  typedef ARG TUPLE;        // typedef allowing for accessing ARG
  enum {SIZE1=ARG::SIZE1}; // enumerator storing the array size

  HOST_DEVICE ExprT(DERIVED* data) : m_exdata(data) {}  // Constructor
  HOST_DEVICE ~ExprT(){}                                // Destructor

  /// Accessor to individual entry
  HOST_DEVICE typename ARG::TYPE at(size_t i) const {return m_exdata->at(i);}
  HOST_DEVICE size_t size() const {return m_exdata->size();} // size
  HOST_DEVICE DERIVED* getData() const {return m_exdata;} // access local data

private:
  DERIVED* m_exdata;   // pointer to local data
};
```

A `HOST_DEVICE` macro is defined so that the code can still compile when CUDA is disabled.

Listing 4: HOST_DEVICE macro definition

```
#ifndef CF_HAVE_CUDA
#define HOST_DEVICE
#else
#define HOST_DEVICE __host__ __device__
#endif
```

Closure object: AddT example. A number of classes corresponding to binary and unary operators are defined as derived classes from `ExprT`. Those are typically indicated as *closure objects*. An example is presented in code listing 5 for the `AddT` class.

Listing 5: AddT class definition

```
template <typename V1, typename V2>
class AddT : public ExprT< AddT<V1,V2>, TPLVEC(V1,V2)> {
public:
  /// Constructor
  HOST_DEVICE AddT (ETYPE(V1) v1, ETYPE(V2) v2) :
     ExprT<AddT<V1,V2>, TPLVEC(V1,V2)>(this), e1(v1), e2(v2) {}

  ///  Accessor to individual entry
  HOST_DEVICE TPLTYPE(V1) at(size_t i) const {return e1.at(i) + e2.at(i);}

  /// Size of the deriving array
  HOST_DEVICE size_t size() const {return e1.size();}

private:
  ETYPE(V1) e1;  // pointer to first operand  expression
  ETYPE(V2) e2; // pointer to second operand expression
};
```

In this case, a few more macros have been introduced to simplify code readability. Those macros are shown in listing 6. Additionally, some basic meta-programming is used to compute the maximum between two integers via the `CMP` class. This is used when parsing a binary operator expression at compile time: the `ETuple` corresponding to its two operands are compared and the maximum `SIZE1` is retained in the resulting expression as the active size. This enables loop unrolling while tackling hybrid expressions including dynamic (or slice) and fixed size arrays.

Listing 6: ET macros definition

```
// simple meta-programming to compared two integers at compile time
template <int N, int M> struct CMP {enum {MAX=(N>M) ? N : M};};
#define NMAX(a,b) CMP<a,b>::MAX

#define TPL(a)     typename a::TUPLE
#define TPLTYPE(a) typename a::TUPLE::TYPE

// ETuple for vectors (M=0) storing the maximum SIZE1 between two operands
// as second template parameter
#define TPLVEC(a,b) ETuple<TPLTYPE(a), NMAX(a::SIZE1, b::SIZE1),0>
```

Please cite this article in press as: A. Lani, et al., A GPU-enabled Finite Volume solver for global magnetospheric simulations on unstructured grids, Computer Physics Communications (2014), http://dx.doi.org/10.1016/j.cpc.2014.06.001

```
// pointers are used instead of references if CUDA compiler is used
#ifndef CF_HAVE_CUDA
#define EREF(a) const a&
#else
#define EREF(a) a
#endif

#define COMA ,
#define ETYPE(a)  EREF(ExprT<a COMA TPL(a)>)
#define ETYPEV(a) EREF(ExprT<a COMA typename a::TUPLE>)
```

Operator overloading functions have to be provided for each closure. In our example for the `AddT` class, the definition of the addition operator overloading is shown in code listing 7 : it only returns an instance of the corresponding closure `AddT`.

Listing 7: Overloading of operator +

```
template <typename V1, typename V2>
HOST_DEVICE  inline AddT<V1,V2> operator+ (ETYPE(V1) v1, ETYPE(V2) v2)
{
  return AddT<V1,V2>(v1,v2);
}
```

Array classes: Vec example. In order to take full profit of the ET capability, array classes deriving from `ExprT` must define a suitable assignment operator accepting a generic expression as argument, as shown in code listings 8 and 9 for the `Vec` class with fixed and dynamic sizes, respectively. In both cases, vector slices `VecSlice`, also implemented as subclasses from `ExprT`, can be created to access just subsets of those arrays. Only fixed size `VecSlice` and fixed size `Vec` have been used on the GPU, since dynamic arrays cannot be allocated on the device.

Listing 8: Vec class definition for fixed size arrays

```
// —— Vec.hh —— //

template <typename T, int N = 0>
class Vec : public ExptrT< Vec<T,N>, ETuple<T,N,0> > {
public:
  /// ... constructor, destructor, member functions, overloading of operators

  /// Overloading of assignment operator
  template <typename EXPR> HOST_DEVICE const Vec<T,N>& operator= (ETYPEV(EXPR) expr)
  {
    for (size_t i=0; i <N; ++i) {m_data[i] = expr.at(i)}
    return *this;
  }

  /// @return a vector slice with fixed size
  template <int NV> HOST_DEVICE VecSlice<T,NV> slice(size_t start);

  /// @return a vector slice whose size is unknown at compile time
  HOST_DEVICE VecSlice<T,0> slice(size_t start, size_t ns);

private:
  T m_data[N]; // array data
};
```

Listing 9: Vec partial specialization for dynamic size arrays

```
// —— Vec.hh —— //

/// the following macro chooses between fixed and dynamic size
#define GETSIZE1(a) ((a>0) ? a : size())
template <typename T>
class Vec<T,0> : ExptrT<Vec<T,0>, ETuple<T,0,0> > {
public:
  /// ... constructor, destructor, member functions, operator overloading, slices

  /// Overloading of assignment operator
  template <typename EXPR> HOST_DEVICE const Vec<T,0>& operator= (ETYPEV(EXPR) expr)
  {
    for (size_t i=0; i < GETSIZE1(NMAX(0,EXPR::SIZE1); ++i) {m_data[i] = expr.at(i)}
    return *this;
  }

private:
  size_t m_size;  // array size
  T*     m_data;  // array data
};
```

Similar concepts to those described here have been applied to implement ET matrices and slices supporting all unary and binary operations (including matrix–matrix products). A key feature of our ET implementation is that one class deriving from an array inherits

Please cite this article in press as: A. Lani, et al., A GPU-enabled Finite Volume solver for global magnetospheric simulations on unstructured grids, Computer Physics Communications (2014), http://dx.doi.org/10.1016/j.cpc.2014.06.001

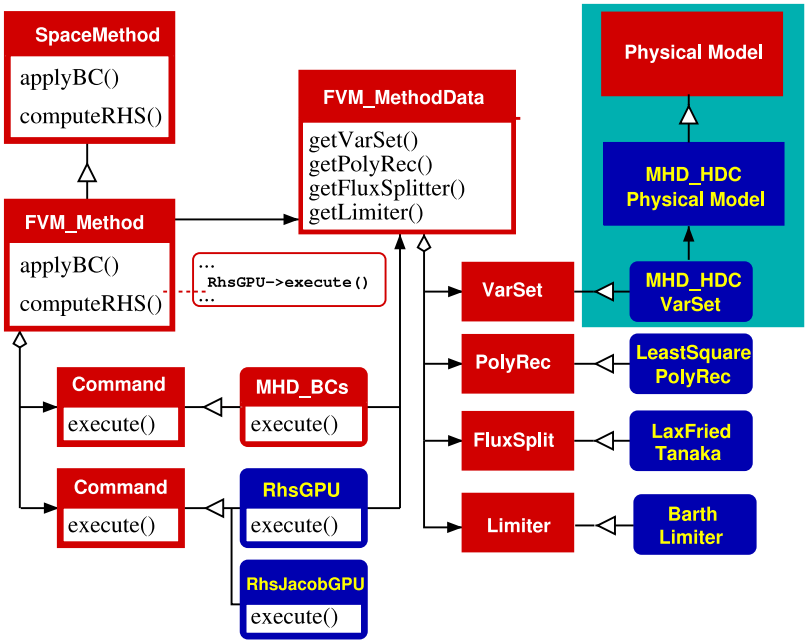

Fig. 3. Schematics of the FV solver for MHD on GPUs. (Red boxes indicate existing objects, while blue boxes stand for new objects implementing the GPU-enabled solver.)

all their semantics and, therefore, can be directly used within mathematical expressions. This means that, as an example, specific **U** or **F** classes can be conveniently defined for implementing Eq. (21).

2.2. FVM–CUDA module

The FVM–CUDA module implements a new generic FV algorithm with CUDA bindings which, for the moment, has been customized for MHD. The overall FV algorithm is decomposed into different components, as depicted in Fig. 3, according to a *Method Command Strategy* (MCS) design pattern (see [6,10] for details) and which is used to implement all numerical modules in COOLFluiD. Herein, red components are part of the core framework and have not been modified during this project, while blue components have been either implemented from scratch (`RhsGPU`, `RhsJacobGPU`) or partially modified (all the others). The abstract interface of `SpaceMethod` is implemented by `FVM_Method` by delegating actions to ad-hoc `Command` objects by calling their *execute*() member function. All commands share access to `FVM_MethodData` which holds pointers to abstract interfaces of algorithms (strategies) such as `PolyRec`, `FluxSplit`, `Limiter`. Dynamic binding to the physics (`MHD_HDC`) is provided by variable sets objects (`VarSet`) [10].

All components of the MCS pattern are *self-registering*, i.e. they can be integrated dynamically and created by name via object factories, and *self-configuring*, i.e. they can configure themselves by fetching their own parameters, even interactively, from the COOLFluiD configuration file. The latter consists of a XML-like list of key-value pairs, where keys are strings giving the path to the parameter to configure and values can be single numbers, strings, arrays, and even arbitrary analytical functions. More details on those powerful techniques and their actual implementation can be found in [6,10,9].

2.2.1. Residual computation

The command `RhsGPU` computes the full RHS on GPU, except for all boundary contributions which are processed separately on the CPU. The implementation of *RhsGPU::execute()* is shown in code listing 10.

Listing 10: *RhsGPU::execute*() implementation

```
// ---- RhsGPU.cu ---- //

initializeComputationRHS(); // initialize the computation of RHS

// convert from polymorphic to static types for the strategy objects
SCHEME* lf   = getMethodData().getFluxSplitter()->dynamic_cast<SCHEME>();
POLYREC* pr  = getMethodData().getPolyReconstructor()->dynamic_cast<POLYREC>();
LIMITER* lm  = getMethodData().getLimiter()->dynamic_cast<LIMITER>();
PHYSICS* phys = getMethodData().getPhysics()->dynamic_cast<PHYSICS>();

// copy parameters belonging to the algorithms and physical model (Strategy objects)
ConfigOptionPtr<POLYREC> dcor(pr);   // parameters for the polynomial reconstruction
ConfigOptionPtr<LIMITER> dcol(lm);   // parameters for the limiter
ConfigOptionPtr<SCHEME>  dcof(lf);   // parameters for the flux scheme
ConfigOptionPtr<PHYSICS> dcop(phys); // parameters for the physics

// copy to GPU solution arrays that change at each iteration
getStateArray()->put();       // copy state vectors array
getGhostStateArray()->put(); // copy ghost states (imposed in dummy cells by BCs)
```

```
// define the number of blocks and threads for the GPU kernel
const int nBlocks =
    CudaEnv::CudaDeviceManager::getInstance().getBlocksPerGrid(nbCells);
const int nThreads =
    CudaEnv::CudaDeviceManager::getInstance().getNThreads();

// define some useful typedefs for simplifying the syntax hereafter
typedef typename SCHEME::template  DeviceFunc<PHYSICS> FluxScheme;
typedef typename POLYREC::template DeviceFunc<PHYSICS> PolyRec;
typedef typename LIMITER::template DeviceFunc<PHYSICS> Limiter;

// compute the cell-based gradients
computeGradientsKernel<PHYSICS, PolyRec> <<<nBlocks,nThreads>>> (dcor.ptr(), ...);

// compute the cell-based limiter
computeLimiterKernel<PHYSICS, PolyRec, Limiter> <<<nBlocks,nThreads>>>
(dcol.ptr(), dcor.ptr(), ...);

// compute the convective flux integral in each cell
computeFluxKernel<FluxScheme, PolyRec> <<<nBlocks,nThreads>>>
(dcof.ptr(), dcor.ptr(), dcop.ptr(),...);

// copy from GPU the data arrays needing for updating the solution
getRhsArray()->get();          // copy RHS
getUpdateCoeffArray()->get();  // copy update coefficient (imposing CFL condition)

finalizeComputationRHS(); // finalize RHS computation
```

After an initialization phase, some polymorphic strategy objects held in `FVM_MethodData` are dynamically cast to their corresponding static types (e.g. `SCHEME`, `POLYREC`, `LIMITER`, `PHYSICS`). All kernels are parameterized (via C++ templates) with GPU-enabled physics and algorithms (reconstruction, limiter, flux scheme) which are defined as nested device functor classes `DeviceFunc` inside the corresponding (pre-existing) concrete strategy objects. An example of this adaptation is shown for the Lax–Friedrichs flux (`LaxFriedTanaka`) in code listing 11.

Listing 11: LaxFriedTanaka definition

```
// ---- LaxFriedTanaka.hh ---- //

class LaxFriedTanaka : public FVMCC_FluxSplitter {
public:

#ifdef CF_HAVE_CUDA /// only activate if CUDA compiler is used
  /// nested class defining local configuration options
  template <typename P = NOTYPE> class DeviceConfigOptions {
  public:
    CFreal coeff; // diffusion reduction coefficient
  };

  /// nested class defining a functor
  template <typename PHYSICS> class DeviceFunc {
  public:
    typedef LaxFriedTanaka BASE;

    /// constructor taking device configuration options as argument
    HOST_DEVICE DeviceFunc(DeviceConfigOptions<NOTYPE>* dco) : m_dco(dco) {}

    /// flux computation
    HOST_DEVICE void operator()(FluxData<PHYSICS>* data, PHYSICS* model);

 private:
    DeviceConfigOptions<NOTYPE>* m_dco; /// pointer to device configuration options
  };

  /// copy the local configuration options to the device
  void copyConfigOptionsToDevice(DeviceConfigOptions<NOTYPE>* dco)
  {
    CFreal coeff = getReductionCoeff();
    cudaMemcpy(&dco->coeff, &coeff, sizeof(CFreal), cudaMemcpyHostToDevice);
  }
#endif

 /// ... the rest of the preexisting LaxFriedTanaka class

}; // end of LaxFriedTanaka class
```

All device functors are parameterized with the actual `PHYSICS`. This design technique relying on static polymorphism allows for accessing the actual number of equations `PHYSICS::NBEQ` (=9 in our case) and dimension of the problem `PHYSICS::DIM` (=3) and enables more aggressive compiler optimization on the device. This behavior remains totally independent from the original definition/implemen-

Please cite this article in press as: A. Lani, et al., A GPU-enabled Finite Volume solver for global magnetospheric simulations on unstructured grids, Computer Physics Communications (2014), http://dx.doi.org/10.1016/j.cpc.2014.06.001

tation of the enclosing class (`LaxFriedTanaka` in our example), which was typically relying on virtual functions and unaware of the actual problem size.

Code listing 11 also shows another nested class, called `DeviceConfigOptions`, which holds some user-defined configuration parameters (from the configuration file) for class `DeviceFunc` to be copied on GPU. `DeviceConfigOptions` accepts one template parameter P, which is set to `NOTYPE` by default.

As shown in code listing 10, suitable `ConfigOptionPtr` objects are created to hold and transparently copy the content of `DeviceConfigOptions` for each strategy object (algorithm and physics) needed by the computational kernels. This is particularly helpful to handle interactive parameters that need to be regularly updated (therefore copied back to the GPU) during the simulation. The class definition of `ConfigOptionPtr` is presented in code listing 12.

Listing 12: ConfigOptionPtr definition

```
// ---- ConfigOptionPtr.hh ---- //
template <typename T, typename P = NOTYPE, DeviceType DT = GPU>
class ConfigOptionPtr {
public:
  /// Constructor
  ConfigOptionPtr(SafePtr<T> obj)
  {
    cudaMalloc((void**)&m_dco, sizeof(T));  // allocate data pointer on device
    obj->copyConfigOptionsToDevice(m_dco);  // copy configurable data to the device
  }

  /// Destructor: deallocates pointer on device
  ~ConfigOptionPtr() {cudaFree(m_dco);}

  /// get the raw pointer
  typename T::template DeviceConfigOptions<P>* ptr() const {return m_dco;}

private:
  typename T::template DeviceConfigOptions<P>* m_dco;  // pointer to the raw data
};
```

After all required data (`CudaVector` arrays or `ConfigOptionPtr`) are copied to the GPU, three kernels are launched in sequence:

- *computeGradientsKernel()* computes the cell gradients feeding the linear reconstruction;
- *computeLimiterKernel()* computes the flux limiter values for each cell;
- *computeFluxKernel()* computes the RHS for all cells.

Both the multi-dimensional linear reconstruction and limiter algorithms are based on cell stencils, therefore it was a straightforward choice to implement them as a parallel cell-based loop in which each cell is mapped onto a GPU thread. The code uses a 1D GPU thread grid, which is a straightforward choice for an unstructured grid in which each cell is associated to a unique ID, and no experiments have been made with 2D or 3D thread grids. The choice was less obvious for the residuals calculation, since, in principle, a face-based loop (as in the original CPU implementation) was also a valid choice. However, different faces contribute to the same cell residual, so that associating a different thread to each face would have required thread synchronization, undoubtedly affecting the overall performance. Hence, we decided to stick to a parallel cell-based loop also for the flux assembly. Since, in our implementation, the cell ordering is kept fully unstructured, it would have been impossible with a single kernel to guarantee access to all needed precomputed gradients and limiters at the exact time they are needed in each thread, even using synchronization. In particular, the computation of each cell limiter relies on precomputed gradients for all cells belonging to the corresponding stencil: this leads naturally to devote a dedicated kernel for precomputing all cell gradients. By using three kernels, gradients and limiter values corresponding to the stencil needed by the flux integration, can all be efficiently precomputed.

Gradient kernel implementation. The kernel is shown in code listing 13. Most of the data arrays stored in global memory are packed inside `KernelData`, while all remaining connectivity information are encapsulated by `CellData`. As shown in the code listing 14, an iterator `CellData::Itr` is used to fetch the data corresponding to the current cell. After having created the concrete `POLYREC` object with its own `DeviceConfigOptions`, the *computeGradients()* defined by the corresponding `DeviceFunc` implements the linear least squares reconstruction presented in Section 1.2.1.

Listing 13: *computeGradientsKernel*() implementation

```
// ---- RhsGPU.cu ---- //

template <typename PHYS, typename POLYREC>
__global__ void computeGradientsKernel
 (typename POLYREC::BASE template DeviceConfigOptions<NOTYPE>* dcor, // parameters
  const CFuint nbCells,        // local total number of cells in this processor
  CFreal* states,              // solution array in the internal cell centers
  CFreal* nodes,               // coordinates of the mesh vertices
  CFreal* centerNodes,         // coordinates of the internal cell centers
  CFreal* ghostStates,         // solution array in the ghost cell centers
  CFreal* ghostNodes,          // coordinates of the ghost cell centers
  CFreal* uX,                  // cell gradients in x
  CFreal* uY,                  // cell gradients in y
  CFreal* uZ,                  // cell gradients in z
  CFreal* limiter,             // cell limiters array
  CFreal* updateCoeff,         // update coefficient imposing CFL condition
```

Please cite this article in press as: A. Lani, et al., A GPU-enabled Finite Volume solver for global magnetospheric simulations on unstructured grids, Computer Physics Communications (2014), http://dx.doi.org/10.1016/j.cpc.2014.06.001

```
  CFreal* rhs,                // cell residuals (flux integrals)
  CFreal* normals,            // face normals
  CFint* isOutward,           // ID of the cell for which normal is outward
  const CFuint* cellInfo,     // storage of useful cell info
  const CFuint* cellStencil,  // stencil connectivity
  const CFuint* cellFaces,    // cell-face IDs connectivity
  const CFuint* cellNodes,    // cell-node IDs connectivity
  const CFint*  neighborTypes, // internal(1), partition(0), boundary(<0) neighbors
  const CellConn* cellConn)   // cell-face, cell-node, face-node connectivities
{
  // each thread takes care of computing the gradient for one single cell
  const int cellID = threadIdx.x + blockIdx.x*blockDim.x;

  if (cellID < nbCells) {
    // all data stored in global memory are packed into KernelData
    KernelData<CFreal> kd (nbCells, states, nodes, centerNodes, ghostStates,
          ghostNodes, updateCoeff, rhs, normals, uX, uY, uZ, isOutward);

    CellData cells(nbCells, cellInfo, cellStencil, cellFaces, cellNodes,
                neighborTypes, cellConn);
    CellData::Itr cell = cells.getItr(cellID);   // create iterator to current cell
    POLYREC polyRec(dcor);   // create polynomial reconstructor
    polyRec.computeGradients(&states[cellID*PHYS::NBEQS],
                             &centerNodes[cellID*PHYS::DIM], &kd, &cell);
  }
}
```

Listing 14: CellData class definition

```
// ---- CellData.hh ---- //

class CellData {
public:
  HOST_DEVICE CellData(/*... all arguments ...*/);  // Constructor
  HOST_DEVICE ~CellData();                          // Destructor

  /// cell iterator
  class Itr {
  public:
    HOST_DEVICE Itr(CellData* cd,  CFuint cellID); // Constructor
    HOST_DEVICE Itr(const CellData::Itr& in)       // Copy constructor
    HOST_DEVICE const CellData::Itr& operator=(const CellData::Itr& in);

    HOST_DEVICE void operator++();                  // Overloading of operator++
    HOST_DEVICE bool operator== (const Itr& other)  // Overloading of the ==
    HOST_DEVICE bool operator!= (const Itr& other); // Overloading of the !=
    HOST_DEVICE bool operator<= (const Itr& other); // Overloading of the <=

    HOST_DEVICE CFuint getStencilSize();              // stencil size
    HOST_DEVICE CFuint getNbFacesInCell() const;      // nb faces in cell
    HOST_DEVICE CFuint getNbActiveFacesInCell() const; // nb active faces in cell,
    HOST_DEVICE CFuint getCellID() const;             // cell ID
    HOST_DEVICE CFuint getShapeIdx() const;           // shape index
    HOST_DEVICE CFuint getNeighborType(CFuint f) const;   // neighbor type
    HOST_DEVICE CFuint getNbFaceNodes(CFuint f) const;    // nodes number in face
    HOST_DEVICE CFuint getNeighborID(CFuint f) const;     // neighbor cell ID
    HOST_DEVICE CFuint getNodeID(CFuint f, CFuint n) const; // ID in given face

  private:
    /// ... member data of CellData::Itr follow
  };

  HOST_DEVICE CellData::Itr getItr(const CFuint cellID); // Get the current cell
  HOST_DEVICE CellData::Itr begin();                     // Get the first cell
  HOST_DEVICE CellData::Itr end();                       // Get the last cell

  /// ... member data of CellData follow
};
```

Limiter kernel implementation. The kernel is shown in code listing 15. After having encapsulated data as in the previous kernel example, the quadrature points for the flux integration on the current cell are calculated and fed to the corresponding *DeviceFunc::limit()* function for computing Barth–Jespersen's limiter as explained in Section 1.2.2. In steady simulations, after a residual drop of a few orders (2–3) of magnitude, the convergence history may tend to oscillate. In order to cure this behavior, our algorithm checks an interactive user-defined residual threshold (`dcor->limitRes`) and, if the current residual (`dcor->currRes`) gets lower than that, applies the *historical modification* introduced by [24]. This consists in choosing the limiter value ϕ_i at the current time step n as:

$$\phi_i^n = \mathtt{min}(\phi_i^{n-1}, \phi_i^n) \tag{22}$$

Please cite this article in press as: A. Lani, et al., A GPU-enabled Finite Volume solver for global magnetospheric simulations on unstructured grids, Computer Physics Communications (2014), http://dx.doi.org/10.1016/j.cpc.2014.06.001

but only after a starting period sufficiently long to obtain a fully developed solution (i.e. with discontinuities located in the right position), during which no special treatment is applied. This treatment is not always needed, but works well for many situations even in the presence of strong shocks.

Listing 15: *computeLimiterKernel*() implementation

```
// ---- RhsGPU.cu ---- //

template <typename PHYS, typename POLYREC, typename LIMITER>
__global__ void computeLimiterKernel(
 (typename LIMITER::BASE::template DeviceConfigOptions<NOTYPE>* dcol,
  typename POLYREC::BASE template DeviceConfigOptions<NOTYPE>* dcor,
  /* same other arguments as in the previous example */)
{
  // each thread takes care of computing the limiter for one single cell
  const int cellID = threadIdx.x + blockIdx.x*blockDim.x;

  if (cellID < nbCells) {
    // compute all cell quadrature points at once
    // (size of this array is overestimated)
    CFreal midFaceCoord[PHYS::DIM*PHYS::DIM*2];

    CellData cells(nbCells, cellInfo, cellStencil, cellFaces,
                   cellNodes, neighborTypes, cellConn);
    CellData::Itr cell = cells.getItr(cellID); // create cell iterator
    const CFuint nbFacesInCell = cell.getNbFacesInCell();
    for (CFuint f = 0; f < nbFacesInCell; ++f) {
      computeFaceCentroid<PHYS>(&cell, f, nodes, &midFaceCoord[f*PHYS::DIM]);
    }

    KernelData<CFreal> kd(/*same arguments as in previous example*/);
    LIMITER limt(dcol);   // create limiter with given configurations

    // if current residual is bigger than threshold compute and store limiter
    if (dcor->currRes > dcor->limitRes)) {
      limt.limit(&kd, &cell, &midFaceCoord[0], &limiter[cellID*PHYS::NBEQS]);
    }
    else {
      // if current residual is less than threshold compute limiter
      CFreal tmpLimiter[PHYS::NBEQS];  // temporary cell limiter array
      limt.limit(&kd, &cell, &midFaceCoord[0], &tmpLimiter[0]); // new limiter
      CFuint currID = cellID*PHYS::NBEQS;
      for (CFuint iVar = 0; iVar < PHYS::NBEQS; ++iVar, ++currID) {
         // apply historical modification of the limiter:
        // store minimum between new and old limiter
        limiter[currID] = min(tmpLimiter[iVar],limiter[currID]);
      }
    }
  }
}
```

Flux kernel implementation. The kernel is shown in code listing 16. Cell residual and update coefficient (i.e. the coefficient imposing the CFL condition) are reset to 0. The latter is defined as

$$k_i = \frac{\mathrm{CFL}}{\sum_f \lambda_f^{+,\max} A_f} \tag{23}$$

where $\lambda_f^{+,\max}$ is the maximum positive wavespeed on face f belonging to cell i, and A_f is the face area. Global data are cached into three objects `KernelData`, `CellData` and `FluxData`. In particular, the latter holds a portion of global data needed for the flux computation (left and right state solution, coordinates, IDs, etc.). All those objects are stored in the thread register memory. Even though the latter might not be an optimal solution, our experiments with trying to store those objects on shared memory instead, using the same structures, provided a much worse performance probably due to the occurrence of bank conflicts which we have not properly taken care of. Future optimization efforts will better address this issue.

A loop over cell faces is performed to (1) compute the quadrature points (face midpoints), (2) extrapolate the solution on those points using the gradients and limiters previously computed, (3) compute the convective fluxes, and (4) compute the face contribution to k_i. In parallel simulations, partition faces (i.e. located on the boundary between different subdomains) are discarded by this loop.

Listing 16: *computeFluxKernel*() implementation

```
// ---- RhsGPU.cu ---- //

template <typename SCHEME, typename POLYREC>
__global__ void computeFluxKernel
 (typename SCHEME::BASE::template DeviceConfigOptions<NOTYPE>* dcof,
  typename LIMITER::BASE::template DeviceConfigOptions<NOTYPE>* dcol,
  typename POLYREC::BASE template DeviceConfigOptions<NOTYPE>* dcor,
  /* same other arguments as in the previous example */)
```

Please cite this article in press as: A. Lani, et al., A GPU-enabled Finite Volume solver for global magnetospheric simulations on unstructured grids, Computer Physics Communications (2014), http://dx.doi.org/10.1016/j.cpc.2014.06.001

```
{
  // each thread takes care of computing the gradient for one single cell
  const int cellID = threadIdx.x + blockIdx.x*blockDim.x;

  if (cellID < nbCells) {
    typedef CudaEnv::VecSlice<CFreal,SCHEME::MODEL::NBEQS> ARRAY;
    ARRAY res(&rhs[cellID*SCHEME::MODEL::NBEQS]);
    res = 0.; updateCoeff[cellID] = 0.;  // reset rhs, update coefficient to 0

    KernelData<CFreal> kd(/*same arguments as in previous example*/);
    POLYREC polyRec(dcor);      // create polynomial reconstructor
    SCHEME fluxScheme(dcof);    // create flux scheme
    typename SCHEME::MODEL pmodel(dcop);  // create physical model

    CFreal midFaceCoord[SCHEME::MODEL::DIM*SCHEME::MODEL::DIM*2];
    FluxData<typename SCHEME::MODEL> currFd;  // create flux data
    currFd.initialize();                      // initialize flux data
    CellData cells(nbCells, cellInfo, cellStencil, cellFaces,
                   cellNodes, neighborTypes, cellConn);
    CellData::Itr cell = cells.getItr(cellID);  // iterator to current cell

    // compute the fluxes on the active neighbors
    // (the first nbFacesInCell are actual cell faces)
    const CFuint nbFacesInCell = cell.getNbActiveFacesInCell();
    for (CFuint f = 0; f < nbFacesInCell; ++f) {
      const CFint stype = cell.getNeighborType(f); // type of neighbor faces

      if (stype != 0) { // skip all partition faces (giving invalid fluxes)
// set all flux data for the current face
const CFuint stateID = cell.getNeighborID(f);
setFluxData(f, stype, stateID, cellID, &kd, &currFd, cellFaces);

// compute face quadrature points (centroid)
CFreal* faceCenters = &midFaceCoord[f*SCHEME::MODEL::DIM];
computeFaceCentroid<typename SCHEME::MODEL>(&cell, f, nodes, faceCenters);

// linearly extrapolate solution on face mid points
polyRec.extrapolateOnFace(&currFd, faceCenters, uX, uY, uZ, limiter);
fluxScheme(&currFd, &pmodel); // convective flux across the face
       ARRAY ress(currFd.getResidual()); // array storing the residual
res -= ress; // update the cell residual
       // update the coefficient imposing CFL condition
updateCoeff[cellID] += currFd.getUpdateCoeff();
      }
    }
  }
}
```

2.2.2. Residual and jacobian computations

The command `RhsJacobGPU` computes the gradients and limiters with the same exact kernels discussed previously, while RHS and its portion of jacobian due to internal faces are calculated by *computeFluxJacobianKernel()*, as shown in code listing 17. In order to overcome the constraint due to the relatively small memory available on the GPU for storing the system sparse matrix and in order, therefore, to be able to deal with problems as large as the CPU can handle, we let the kernel fill the matrix gradually, considering only a few block rows at the time. The number of blocks to launch on the kernel is given by `m_nbKernelBlocks` for which a value of 64 appeared to work best in our case. The number of cells associated to each kernel launch is precomputed and stored into an array `m_nbCellsInKernel`. All jacobian contributions from boundary faces are processed separately by *executeBC* () on CPU. The existing boundary conditions (`MHD_BCs` objects in Fig. 3) have not been modified and their processing is fully computed on the CPU. This choice, while not critical for performance since the computation of boundary fluxes and flux Jacobians has a negligible cost if compared to the rest, is of crucial importance for maintaining the flexibility and relative simplicity of the code.

Listing 17: *RhsJacobGPU::execute*() implementation

```
// ——— RhsJacobGPU.cu ——— //

// ... same as RhsGPU (initialization, kernels for gradients and limiter)

// compute the flux jacobian in each cell
ConfigOptionPtr<NumericalJacobian, PHYSICS, GPU> dcon
   (&this->getMethodData().getNumericalJacobian());

CFuint startCellID = 0;
for (CFuint s = 0; s < m_nbCellsInKernel.size(); ++s) {
  computeFluxJacobianKernel<FluxScheme, PolyRec, Limiter>
     <<<nbKernelBlocks, nThreads>>>
(dcof.ptr(), dcor.ptr(), dcol.ptr(), dcon.ptr(), dcop.ptr(), ...);

  // copy from GPU the block matrix portion filled in by the kernel
```

Table 1
Initial ACE conditions for the solar wind/Earth's magnetosphere interaction case.

ρ_∞	$V_{x\infty}$	$V_{y\infty}$	$V_{z\infty}$	$B_{x\infty}$	$B_{y\infty}$	$B_{z\infty}$	p_∞
1.26020	−10.8434	−0.859678	0.0146937	0.591792	−2.13282	−0.602181	0.565198

Table 2
Timings of the different steps in the residual computation for one call to `execute()`.

	explicit (coarse) (%)	explicit (fine) (%)	implicit (coarse) (%)	implicit (fine) (%)
CPU–>GPU transfer	0.5	0.5	0.1	0.1
gradients kernel	60.7	60.7	10.7	10.2
limiter kernel	8.4	8.4	1.5	1.4
flux/jacobian kernel	29.7	29.9	87.6	88.2
GPU–>CPU transfer	0.7	0.5	0.1	0.1

```
  m_blockJacobians.get();

  // update the portion of system matrix computed by this kernel
  updateSystemMatrix(s);

  // update the counter keeping track of the starting cell ID
  startCellID += m_nbCellsInKernel[s];
}

// compute flux jacobians on boundaries
executeBC();

// ... same as RhsGPU (finalization)
```

3. Numerical results and benchmarks

3.1. Application: 3D solar wind/ Earth's magnetosphere interaction

For our numerical benchmarks, we have chosen a representative and challenging 3D simulation involving multi-physical modeling of complex behavior of plasma originating from the Sun, and travelling the interplanetary space between the Sun and the Earth (i.e. *solar wind*). The main aim of this testcase is to investigate the effects of the interaction between the solar wind and the near-Earth plasma environment (i.e. *Earth's magnetosphere*). This constant interaction can sometimes be more violent than average during the so-called *geomagnetic storms*, causing severe damages on the space-borne and ground-based technological systems or even on human health. Therefore, the prediction of geomagnetic storms is an important issue. In this section, we will discuss the code performance on a 3D global MHD simulation with conditions corresponding to the geomagnetic storm that occurred on April 6th, 2000. The driving mechanism for this unsteady simulation is the time-varying measurements of the solar wind plasma parameters by the NASA Advanced Composition Explorer (ACE) satellite. For prediction purposes, this type of computations should be performed faster than the real physical time, therefore necessitating as efficient parallel processing algorithms as possible.

3.2. Testcase definition

The computational domain is a rectangular box with dimensions of $-200 \leq x \leq 235$ and $-50 \leq y, z \leq 50$ including a hollow sphere of radius 2.5, centered in the origin. The coordinate system utilized is Geocentric Solar Magnetospheric (GSM) for which the x-axis points at the Sun and the xz-plane contains the dipole axis where a fixed tilt angle of 11.94° is applied. The boundary conditions are superfast inlet and outlet, and ionosphere/magnetosphere interaction. At the superfast inlet (located at $x = 235$), the solar wind plasma parameters extracted from the 64 [s]-average ACE satellite data are imposed. The superfast outlet is applied to the rest of the box surfaces where the plasma parameters are simply extrapolated to boundary ghost cells. Finally, the boundary condition suggested in [38,39] is applied at the ionosphere/magnetosphere interaction boundary. Accordingly, the plasma number density is fixed to 56 AMU/cm^3 as suggested in [40], the temperature is fixed to 35,000 K, a no-slip condition is imposed for the velocity and a mirror boundary condition is applied for $\mathbf{B}_1$. A steady simulation is performed by imposing at the superfast inlet the solar wind plasma parameters measured by the ACE satellite at a certain instant before the geomagnetic storm, corresponding to the non-dimensional free stream conditions in Table 1.

Two fully unstructured meshes, with 976,344 (coarse) and 1,632,885 tetrahedral cells (fine) respectively, have been used for our numerical experiments. A global view of the computational domain and a zoomed view showing the finer mesh resolution on the symmetry planes and on the Earth surface are shown in Fig. 4. Fig. 5 shows additional details of the grid on the xz and $x = -30$ planes.

3.3. Performance results

The performance of the GPU-enabled code has been tested with different time integrators and number of CPUs/GPUs in the following cases:

- explicit steady with 1-stage Runge–Kutta (i.e. Forward Euler);
- explicit unsteady with 4-stage Runge–Kutta;

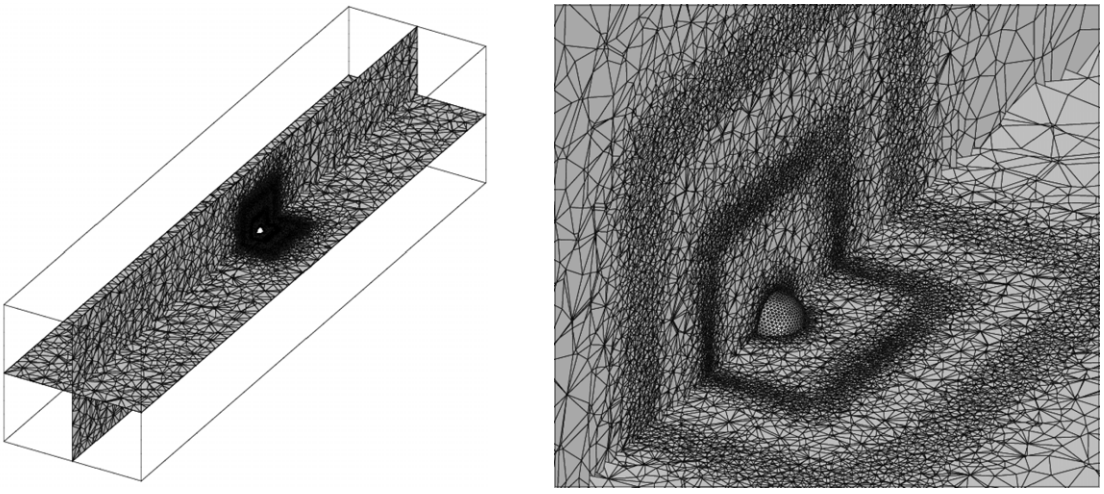

Fig. 4. Computational domain with views on the symmetry planes and sphere surface mesh.

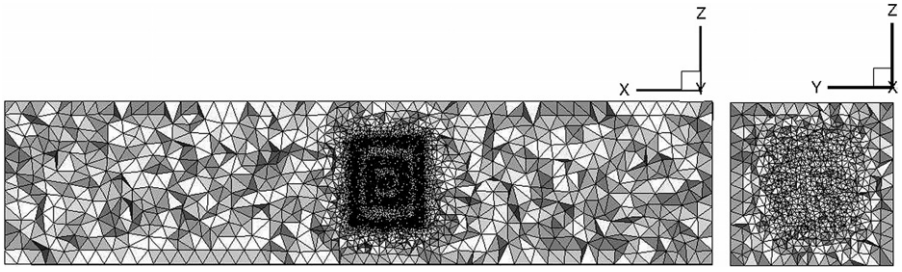

Fig. 5. The grid in the xz and $x = -30$ planes.

Fig. 6. One of the NVIDIA Tesla K10 GPU accelerators which has been used for our tests.

- implicit steady with 1-point Backward Euler, GMRES solver and Additive Schwartz preconditioner (KSPGMRES and PCASM in PETSc);
- implicit unsteady with 3-point Backward Euler, GMRES solver and Additive Schwartz preconditioner (KSPGMRES and PCASM in PETSc).

All benchmarks to be presented have been run on a dual 6-core Intel(R) Xeon(R) CPU E5-2640, 2.50 GHz, 132 Gb of RAM, featuring four NVIDIA Tesla K10 GPU accelerators (see Fig. 6), featuring each two GK104 GPUs (i.e. 8 GPUs in total). For each test we have considered the total wall time corresponding to three time steps (only accounting for the iteration time, without including any pre- or post-processing).

3.3.1. Residual computation on GPU

Table 2 shows the contributions of the different computational steps to the total run time for one call to the `execute()` function (i.e. the core of the residual/jacobian computation) in the explicit and implicit case on both coarse and fine meshes. Only a single GPU has been used for this benchmark. While in the explicit case the least square reconstruction takes the most of the computational time (60% of the total), in the implicit case the flux+jacobian assembly prevails (88%). No noticeable difference exist between coarse and fine meshes.

Please cite this article in press as: A. Lani, et al., A GPU-enabled Finite Volume solver for global magnetospheric simulations on unstructured grids, Computer Physics Communications (2014), http://dx.doi.org/10.1016/j.cpc.2014.06.001

Table 3
Performance on a coarse unstructured mesh (976,344 tetrahedral cells).

	1 GPU vs 1 CPU	8 GPUs vs 8 CPUs	8 GPUs vs 24 CPUs	8 vs 1 GPU
explicit steady	11.47X	8.94X	7.28X	4.99X
explicit unsteady	10.77X	9.01X	6.75X	5.18X
implicit steady	2.71X	2.54X	1.61X	6.36X
implicit unsteady	1.59X	1.52X	1.35X	6.35X

Table 4
Performance on a fine unstructured mesh (1,632,885 tetrahedral cells).

	1 GPU vs 1 CPU	8 GPUs vs 8 CPUs	8 GPUs vs 24 CPUs	8 vs 1 GPU
explicit steady	12.32X	8.63X	6.76X	5.21X
explicit unsteady	10.07X	9.25X	7.09X	5.8X
implicit steady	2.37X	2.55X	1.74X	6.94X
implicit unsteady	1.4X	1.51X	1.11X	7.0X

Moreover, in all cases the data transfer time is negligible compared to the rest (up to 1% in explicit and 0.2% in implicit cases). On the one hand, this is a positive result, meaning that our algorithm is not bound by it. On the other hand, this also indicates that there is probably a lot of room for algorithmic optimization of the kernel, since typically data transfer ends up being the bottleneck of highly optimized implementations on GPU.

3.3.2. Comparative performance between GPU and CPU

Tables 3 and 4 show the results in terms of relative performances on the two meshes. In all performance results, when referring to one "GPU", we mean one GPU plus the corresponding CPU core. We compare the relative speedups between 1 GPU and 1 CPU, 8 GPUs and 8 CPUs, and 8 GPUs and 24 CPUs. This last test was considered because 24 CPU-cores gave the best achievable performance on our machine without using GPUs, thanks to the hyper-threading technology. In multi-CPU/GPU runs, data corresponding to the overlap regions shared by mesh partitions is exchanged via MPI among different nodes during a synchronization phase. The latter follows each time step in steady simulations or each sub-iteration within each time-step in unsteady simulations.

Those results show that one GPU is up to 12.3X faster than one CPU for explicit calculations, steady or unsteady, up to 2.7X and 1.6X for implicit steady and unsteady calculations, respectively. The situation is very similar if 8 GPUs are compared against 8 CPUs. The code could not be tested on more GPUs since in our current implementation we assume that one CPU-core is mapped onto one GPU and 8 was the maximum number of GPUs we had access to. If we compare the best machine performance using all possible (24) CPU-cores with hyperthreading, the 8-GPU performance is still 7X faster for explicit and up to 70% faster for implicit cases. Despite this last comparison being rather unfavorable for the GPUs, the performance gain provided by GPUs is still appreciable. Moreover, the code scales well on multiple GPUs, providing a parallel efficiency (computed as the ratio between the speedup given by "8 GPU vs. 1 GPU" and 8, i.e. the maximum number of GPUs) up to 88% for the implicit cases and up to 73% for the explicit ones.

To further assess the performance, we ran a full steady implicit calculation on the coarse and fine grids from scratch till convergence with 8 GPUs and then with 8 CPUs. The GPU-enabled code gave a speedup of 2.22 and a total wall time of 1,007 s on the coarse mesh, a speedup of 1.66 and a total wall time of 7,467 s on the fine mesh. A final unsteady implicit calculation was restarted from the converged steady solution on the fine mesh and fed by ACE data for 67,543 s (i.e. 18 h and 45 min) of physical time corresponding to 1200 time steps. In this case, the simulation ran 1.5X faster on 8 GPUs than on 8 CPUs, for a total wall time of 185,326 s (i.e. 51 h and a half). Those last real life, endurance tests confirmed the reliability of the corresponding benchmark results in Table 4, which were based on only 3 time steps. The moderate decrease in performance for the full steady cases can be explained with an averagely larger number of Krylov sub-iterations per time step needed by PETSc. A reasonable value of 50 was set as maximum number for those sub-iterations.

3.4. Numerical solutions

Numerical results for the steady case on the fine mesh are shown in Figs. 7 and 8 (left) in terms of plasma number density contours and magnetic field lines in the vicinity of the Earth, respectively. The convergence history is presented in Fig. 8 (right) in terms of density (ρ) residual. The numerical scheme is initially first order accurate and then the effect of the second-order reconstruction is injected gradually (in 2 steps) by using an interactive parameter (in the range [0,1] where 0 corresponds to the first order scheme and 1 corresponds to the application of the full linear reconstruction) in front of the limited gradient term for each conservative solution variable. As a result, the convergence history shows a peak at each increase in the value of this interactive parameter. The first peak in Fig. 8 (right), however, corresponds to the increase in the CFL number from the initial value of 10 to its final value of 10^6.

The result of the steady simulation is used as an initial condition for the unsteady run. The time-step is imposed to be 56 s, while the maximum number of sub-iterations per time-step is set to 5 for our benchmarks. Also in this case, results are presented only for the fine mesh. A physical time of 18 h and 45 min is simulated in 22 h and 54 min on 60 CPUs. This result looks promising for the prediction capability of the solver as the storm can be simulated in faster than the real physical time on this grid with only a slight increase in the number of processors. Figs. 9 and 10 show the plasma number density contours and magnetic field lines in the vicinity of the Earth at different instants during the geomagnetic storm, respectively.

4. Conclusion

A detailed description of a parallel GPU-enabled FV solver for ideal MHD equations in complex 3D steady/unsteady problems has been provided in this paper. After a detailed description of the governing equations and of the state-of-the-art numerical method, the attention has been focused on some innovative object-oriented design and implementation aspects related to the porting to GPUs, which are

Please cite this article in press as: A. Lani, et al., A GPU-enabled Finite Volume solver for global magnetospheric simulations on unstructured grids, Computer Physics Communications (2014), http://dx.doi.org/10.1016/j.cpc.2014.06.001

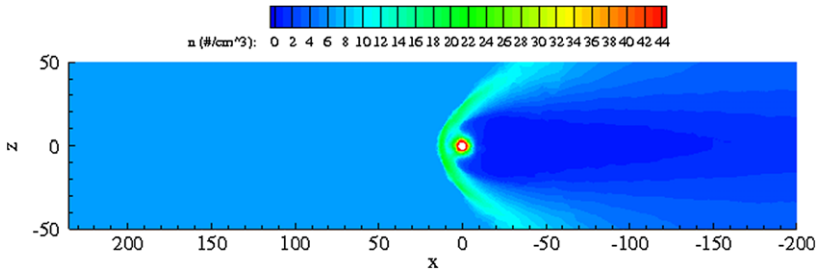

Fig. 7. Plasma number density contours in the *xz*-plane for the converged steady simulation.

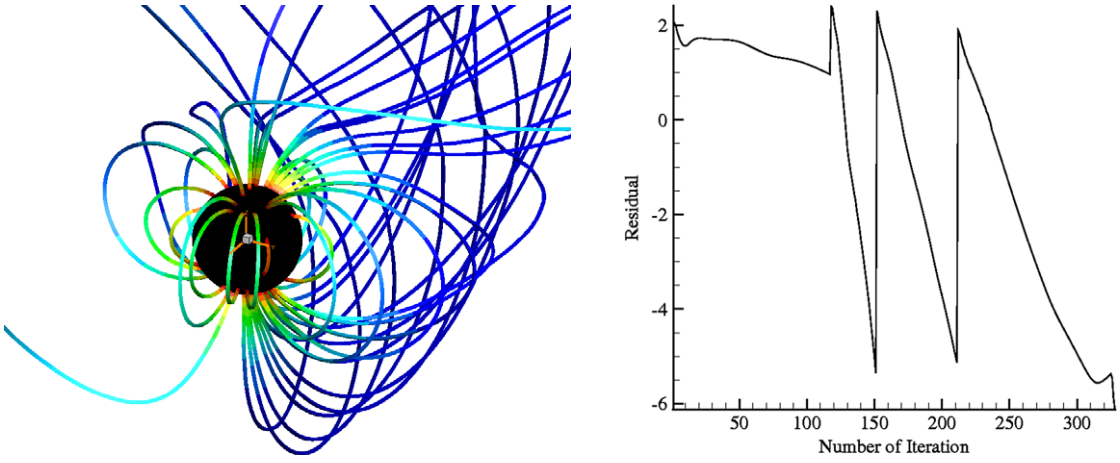

Fig. 8. Magnetic field lines in the vicinity of the Earth (left) and convergence history in terms of density residual (right).

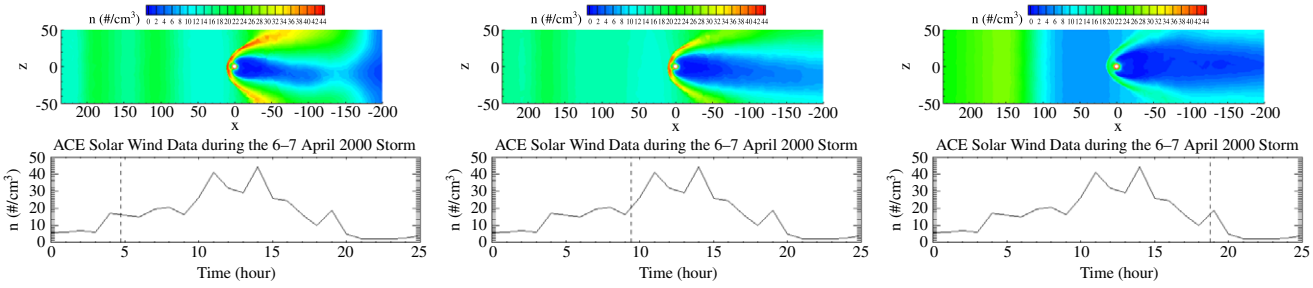

Fig. 9. Plasma number density contours in the *xz*-plane at 3 instants during the 04/06/2000 geomagnetic storm together with the inlet solar wind number density data measured by the ACE satellite and the location of the instants indicated by vertical dashed lines.

certainly reusable and can be inspirational for other applications as well. Despite an incomplete code optimization which will be subject of future work, the performance boost of the resulting code is already appreciable and results are promising. While the performance for cases with explicit time stepping totally relies on our implementation, the PETSc [28] linear system solver seems to represent the bottleneck for the performance of implicit calculations. Particularly, if the number of Krylov sub-iterations in the GMRES solver is larger than 20–30 (it can be easily $\geq$ 50 at high CFL numbers or in unsteady simulations), most of the computational time is spent inside PETSc. Even if not discussed explicitly in the paper, a recent GPU-enabled version of PETSc was interfaced and tested by us with the hope of further improving the performance of our code. At the time of the writing, despite the availability of a GPU-enabled GMRES solver, only Jacobi and algebraic multigrid preconditioners are supported on multiple GPUs. Unfortunately, none of those preconditioners is able to converge our stiff convection-dominated cases. PETSc developers are currently in the process of porting to GPU more sophisticated preconditioners (Additive Schwartz and block Jacobi) which should suit our needs and provide a significant boost in performance (possibly up to 2X).

To conclude, our main focus in this work has been to develop and apply advanced object-oriented coding techniques to port an existing state-of-the-art MHD code to run on GPUs, while keeping a overall flexible and clean design. After having invested some effort in trying to improve the resulting performance, which is noticeably better than the native CPU-based code, we have decided to postpone any fine tuning to a second phase. In our early optimization attempts, while trying to improve memory coalescing, which is known to be not a trivial task for unstructured meshes, we reorder the cells with a Reverse Cuthill McKee ordering algorithm. This experiment only lead to a 5% speed up on the steady explicit case on a single-GPU, therefore it was decided not to invest more effort in applying it to a parallel case which would have lead to an overly complicated algorithm with very little benefit. In any case, due to the large stencil needed by the multidimensional linear reconstruction algorithm, which is the most critical portion of the code on the GPU for explicit runs, an optimal coalescing would not necessarily improve performance since different threads would also have a higher probability to access the same global data (belonging to the stencil of neighboring cells/threads) simultaneously, which could hamper parallelism. We believe that a better reordering of the cells (perhaps based on a red-black type of algorithm) and a better use of shared memory could lead up to 5X or maybe

Please cite this article in press as: A. Lani, et al., A GPU-enabled Finite Volume solver for global magnetospheric simulations on unstructured grids, Computer Physics Communications (2014), http://dx.doi.org/10.1016/j.cpc.2014.06.001

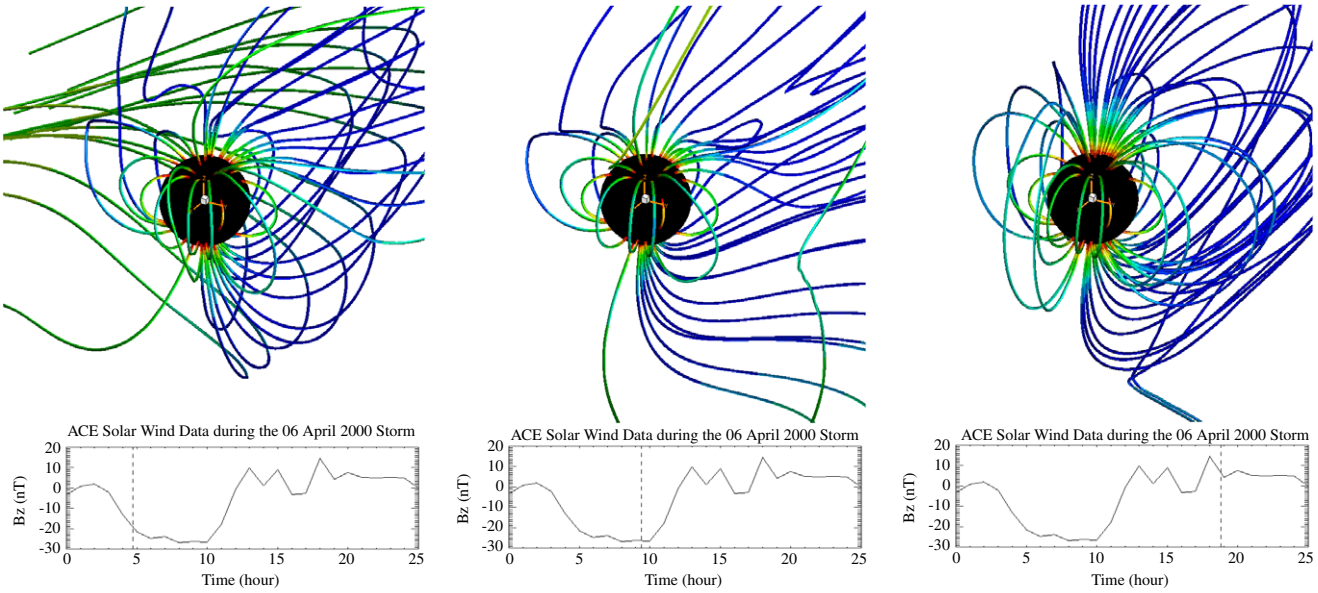

Fig. 10. Magnetic field lines in the vicinity of the Earth at 3 instants during the 04/06/2000 geomagnetic storm together with the inlet solar wind z-component of the ACE magnetic field data and the location of the instants indicated by vertical dashed lines.

even 10X speed up on top of the current performance for explicit cases. Additional benefits may come from the use of GPUDirect instead of MPI [2], but this would also require substantial changes in the existing parallel infrastructure of COOLFluiD which are not foreseen in the near future. Luckily, minimizing data transfer does not seem to be an issue in our current implementation where 99% of the run-time is spent inside the kernels.

Acknowledgments

The authors acknowledge the financial support from the C 90347 (ESA PRODEX 10) project and from the projects GOA/2009-009 (KU Leuven) and G.0729.11 (FWO Vlaanderen). For some simulations, the authors used the infrastructure of the VSC "Flemish Supercomputer Center", funded by the Hercules foundation and the Flemish Government, department EWI.

References

[1] H.C. Wong, U. Wong, X. Feng, Z. Tang, Comput. Phys. Commun. 182 (2011) 2132–2160.
[2] U.H. Wong, H.C. Wong, Y. Ma, Comput. Phys. Commun. 185 (2014) 144–152.
[3] M.S. Yalim, D.V. Abeele, A. Lani, Proc. of the 11th International Conference on Hyperbolic Problems, Ecole Normale Supérieure, Springer-Verlag, Lyon, France, 2006, pp. 1085–1092.
[4] M.S. Yalim, D.V. Abeele, A. Lani, T. Quintino, H. Deconinck, J. Comput. Phys. 230 (15) (2011) 6136–6154.
[5] A. Lani, T. Quintino, D. Kimpe, H. Deconinck, S. Vandewalle, S. Poedts, in: V.S. Sunderan, G.D. van Albada, P.M.A. Sloot, J.J. Dongarra (Eds.), Computational Science ICCS 2005, in: LNCS 3514, Vol.1, Emory University, Springer, Atlanta, GA, USA, 2005, pp. 281–286.
[6] A. Lani, T. Quintino, D. Kimpe, H. Deconinck, S. Vandewalle, S. Poedts, Scientific Programming. 14 (2) (2006) 111–139. Special Edition on POOSC 2005.
[7] D. Kimpe, A. Lani, T. Quintino, S. Poedts, S. Vandewalle, in: D.K. B. Di Martino, J.J. Dongarra (Eds.), Proc. 12th European Parallel Virtual Machine and Message Passing Interface Conference, Springer, Sorrento, 2005, pp. 520–527.
[8] A. Lani, N. Villedieu, K. Bensassi, L. Kapa, M. Vymazal, M.S. Yalim, M. Panesi, in: AIAA 2013-2589, 21th AIAA CFD Conference, San Diego (CA), 2013.
[9] T. Quintino, A Component Environment for High-Performance Scientific Computing. Design and Implementation, (Ph.D. thesis), Katholieke Universiteit Leuven, 2008.
[10] A. Lani, An Object Oriented and High Performance Platform for Aerothermodynamics Simulation, (Ph.D. thesis), Université Libre de Bruxelles, 2008.
[11] M. Panesi, A. Lani, T. Magin, F. Pinna, O. Chazot, H. Deconinck, in: AIAA Paper 2007-4317, 38th AIAA Plasmadynamics and Lasers Conference, Miami (Florida), 2007.
[12] G. Degrez, A. Lani, M. Panesi, O. Chazot, H. Deconinck, J. Phys. D: App. Phys 41 (2009).
[13] D. Knight, J. Longo, D. Drikakis, D. Gaitonde, A.L., et al., Prog. Aerosp. Sci. 48-49 (2012) 8–26.
[14] A. Munafo, A. Lani, A. Bultel, M. Panesi, Phys. Plasma 20 (7) (2013).
[15] M. Panesi, A. Lani, Phys. Fluids 25 (2013) 057101.
[16] A. Lani, P.D. Santos, A. Sanna, in: AIAA 2013-2893, 44th AIAA Thermophysics Conference, San Diego (CA), 2013.
[17] G. Toth, J. Comput. Phys. 161 (2000) 605–652.
[18] A. Dedner, F. Kemm, D. Kroner, C.-D. Munz, T. Schnitzer, M. Wesenberg, J. Comput. Phys. 175 (2002) 645–673.
[19] T. Tanaka, J. Comput. Phys. 111 (1994) 381–389.
[20] M.S. Yalim, An artificial compressibility analogy approach for compressible ideal mhd: application to space weather simulation, (Ph.D. thesis), Université Libre de Bruxelles, 2008.
[21] T. Tanaka, J. Geophys. Res. 100 (A7) (1995) 12057–12074.
[22] T. Barth, 25th Computational Fluid Dynamics Lecture Series, Von Karman Institute, 1994.
[23] T. Barth, D. Jespersen, The design and application of upwind schemes on unstructured meshes, Tech. rep., AIAA, technical Report 89-0366 (1989).
[24] M. Delanaye, Polynomial reconstruction finite volume schemes for the compressible Euler and Navier–Stokes equations on unstructured adaptive grids, (Ph.D. thesis), University of Liege, Faculty of Applied Sciences, 1996.
[25] I. Lepot, A parallel high-order implicit finite volume method for three-dimensional inviscid compressible flows on deforming unstructure meshes, (Ph.D. thesis), University of Liege, Faculty of Applied Sciences, 2004.
[26] J.A. Trangenstein, Numerical Solution of Hyperbolic Partial Differential Equations, first ed., Cambridge University Press, Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, Sao Paulo, 2009.
[27] P. Roe, J. Comput. Phys. 43 (1981) 357–372.
[28] A.N. Laboratory, Petsc: Portable, extensible toolkit for scientific computation, http://www-unix.mcs.anl.gov/petsc (2007).
[29] Thrust, http://thrust.github.io (2014).
[30] T. Veldhuizen, C++ Report 7 (5) (1995) 26–31.

Please cite this article in press as: A. Lani, et al., A GPU-enabled Finite Volume solver for global magnetospheric simulations on unstructured grids, Computer Physics Communications (2014), http://dx.doi.org/10.1016/j.cpc.2014.06.001

[31] T. Veldhuizen, C++ Report 7 (4) (1995) 36–43.
[32] D. Vandevoorde, N.M. Josuttis, C++ Templates, Addison-Wesley, 2003.
[33] Eigen, http://eigen.tuxfamily.org/index.php?title=Main_Page (2014).
[34] E.N., et al. Boost proto http://www.boost.org/doc/libs/1_55_0/doc/html/proto.html (2008).
[35] O. Petzold, Tvmet: Tiny vector matrix library using expression templates, http://tvmet.sourceforge.net (2014).
[36] K.R., et al. ViennaCL - linear algebra library using cuda, opencl and openmp, http://viennacl.sourceforge.net (2014).
[37] A. Lani, H. Deconinck, POOSC'06 Workshop Notes, Von Karman Institute, Nantes, 2006.
[38] K.G. Powell, P.L. Roe, T.J. Linde, T.I. Gombosi, D.L. De Zeeuw, J. Comput. Phys. 154 (1999) 284–309.
[39] T.I. Gombosi, G. Toth, D.L. De Zeeuw, K.C. Hansen, K. Kabin, K.G. Powell, J. Comput. Phys. 177 (2002) 176–205.
[40] A.J. Ridley, T.I. Gombosi, I.V. Sokolov, G. Toth, D.T. Welling, Ann. Geophys. 28 (2010) 1589–1614.

Please cite this article in press as: A. Lani, et al., A GPU-enabled Finite Volume solver for global magnetospheric simulations on unstructured grids, Computer Physics Communications (2014), http://dx.doi.org/10.1016/j.cpc.2014.06.001